Supporting requirements update during software evolution

Ben Charrada, Eya ; Koziolek, Anne ; Glinz, Martin

Abstract: Updating the requirements specification when software systems evolve is a manual task that is expensive and time consuming. Therefore, maintainers usually apply the changes to the code directly and leave the requirements unchanged. This results in the requirements rapidly becoming obsolete and useless. In this paper, we propose an approach that supports the maintainer in keeping the requirements specification consistent with the implementation, by identifying the requirements that are impacted whenever the code is changed. Our approach works as follows. First, we analyze the changes that have been applied to the source code and detect if they are likely to impact the requirements or not. Second, we trace the requirements-impacting changes back to the requirements specification to identify the parts that might need to be modified. The output of the tracing is a list of requirements that are sorted according to their likelihood of being impacted. Automatically identifying the parts of the requirements specification that are likely to need maintenance reduces the effort needed for keeping the requirements up-to-date and thus makes the task of the maintainer easier. When applying our approach in three cases studies, 70% to 100% of the impacted requirements were identified within a list that includes less than 20% of the total number of requirements in the specification.

DOI: https://doi.org/10.1002/smr.1705

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2015; **27**:166–194
Published online 20 March 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1705

Supporting requirements update during software evolution

Eya Ben Charrada[1,*,†], Anne Koziolek[2] and Martin Glinz[1]

[1]*Department of Informatics, University of Zurich, Zurich, Switzerland*
[2]*Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, Germany*

SUMMARY

Updating the requirements specification when software systems evolve is a manual task that is expensive and time consuming. Therefore, maintainers usually apply the changes to the code directly and leave the requirements unchanged. This results in the requirements rapidly becoming obsolete and useless. In this paper, we propose an approach that supports the maintainer in keeping the requirements specification consistent with the implementation, by identifying the requirements that are impacted whenever the code is changed. Our approach works as follows. First, we analyze the changes that have been applied to the source code and detect if they are likely to impact the requirements or not. Second, we trace the requirements-impacting changes back to the requirements specification to identify the parts that might need to be modified. The output of the tracing is a list of requirements that are sorted according to their likelihood of being impacted. Automatically identifying the parts of the requirements specification that are likely to need maintenance reduces the effort needed for keeping the requirements up-to-date and thus makes the task of the maintainer easier. When applying our approach in three cases studies, 70% to 100% of the impacted requirements were identified within a list that includes less than 20% of the total number of requirements in the specification.

Received 21 November 2014; Accepted 4 January 2015

KEY WORDS: requirements evolution; requirements update; impact analysis; traceability; artifact synchronization

1. INTRODUCTION

When maintaining and evolving software, an up-to-date requirements specification provides much knowledge about the software that is very useful for supporting several maintenance and evolution tasks. For example, the requirements specification includes the rationale behind the implementation, which can support program comprehension and which also prevents undoing important decisions by accident. Additionally, the requirements are usually written in natural language and thus can be used to discuss changes with stakeholders who are not from the software engineering domain. Therefore, if the information contained in the requirements specification becomes outdated and unreliable, the maintainability of the software system will be hindered, and the system will eventually enter the *servicing stage* [1] where only minor changes can be applied to it.

In practice, however, requirements are usually not updated when software systems evolve [1, 2]. This is mainly because updating requirements is still a manual task that is very expensive and time consuming. In fact, the maintainer has to go through the whole requirements specification, which can include hundreds or thousands of pages, and find the parts that need to be changed. Therefore, maintainers usually apply changes to the code directly, but do not update the requirements

*Correspondence to: Eya Ben Charrada, Department of Informatics, University of Zurich, Zurich, Switzerland.
†E-mail: charrada@ifi.uzh.ch

specification as observed by Lethbridge *et al*. [2], for example. Consequently, the requirements specification rapidly becomes obsolete and useless.

The goal of this work is to support the maintainer in keeping the requirements specification up-to-date when software systems evolve. We propose a new technique that automatically identifies the requirements that are likely to be impacted whenever the source code is changed. Our approach detects the source code changes that are likely to impact the requirements and traces these changes back to the requirements specification in order to identify the parts that might require maintenance. The approach is automated and does not require any manually created and maintained traces. Our approach is applicable to functional requirements and requirements that are specified in an operational representation [4], which together are the dominant requirements in most systems.

To evaluate our approach, we applied it in three case studies: AquaLush, iTrust, and Connect. For all case studies, our approach succeeded to detect most of the code changes that impact the requirements while ignoring irrelevant changes such as bug fixes and refactoring. When tracing the changes to the requirements specification, our approach identified most or all impacted requirements while filtering out more than 80% of the non-impacted ones for all case studies. We expect our approach to encourage maintainers to update the requirements specification regularly, as it reduces the number of requirements the maintainer has to look at during the update.

This paper is an extension of an existing conference paper [5]. In this extension, we have three new contributions. First, we changed the scope of our approach: before, the approach was meant to be used after each code release, but now it is used after each commit (or evolution task [6]). Second, we have extended our tracing technique so that it merges the traces obtained from various classes in one final ranked list using a new scoring technique. Finally, we added two additional evaluations of the approach on two new case studies (AquaLush and Connect), and we re-evaluated the new tracing technique (with the scoring) on the iTrust case study, which was used in [5].

The rest of the paper is organized as follows. We define the terms *outdated*, *impacted*, and *detected*, which are frequently used in this article, in Section 2. In Section 3, we give an overview of the approach and its usage. In Section 4, we present the first step of the approach, which aims at identifying whether the changes of a commit are requirements-impacting or not. In Section 5, we present the second part of the approach, which consists of extracting the keywords for each change and tracing them to the requirements. The tool implementing our approach is introduced in Section 6. In Section 7, we present the evaluation of our approach in three case studies: AquaLush, iTrust, and Connect. We discuss the results of our approach in Section 8 and the related work about requirements update and software traceability in Section 9. The conclusion and future work are presented in Section 10.

2. OUTDATED, IMPACTED, AND DETECTED REQUIREMENTS

The terms *outdated* requirement, *impacted* requirement, and *detected* requirement are frequently used in this article. In this section, we define each of these terms. We consider a requirement in the requirements specification to be *outdated* if it no longer reflects the current needs of the stakeholders. When maintainers make a change to the source code such that a requirement or a group of requirements in the requirements specification become inconsistent with the new version of the source code, then this requirement or group of requirements are *impacted* by the change. When the change was carried out to comply with new or changed stakeholder needs, the impacted requirements are also outdated. However, the two categories do not fully overlap. For example, a requirement can become outdated because of the changing stakeholder needs, but it is not impacted because the source code is not (or not yet) changed. Conversely, making changes to the source code without a prior impact analysis can easily impact requirements, although these are not outdated. However, in a well-managed software evolution process, the source code is usually well aligned with the stakeholders' needs such that the outdated requirements are covered by the impacted requirements to a large extent. Figure 1 illustrates this situation. The goal of this work is to help the maintainer identify the outdated requirements in the requirements specification by automatically detecting the impacted requirements whenever the source code is changed. These *detected* requirements are the requirements that our approach detects and which the maintainer will need to look at in order to update the requirements specification. Ideally, all

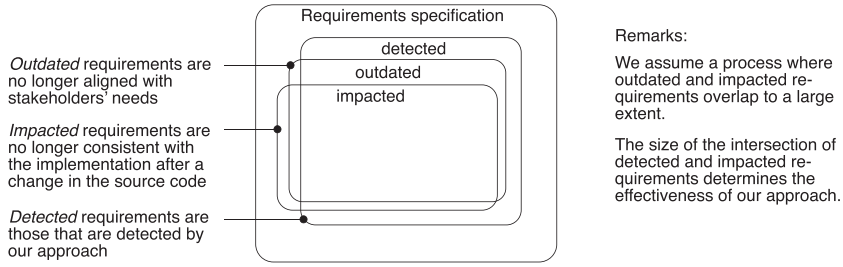

Figure 1. Outdated, impacted, and detected requirements.

impacted requirements would be detected. However, as our approach includes automatic classification and tracing, there will always be false positives (detected requirements that are not impacted) and false negatives (impacted requirements that are not detected); see Figure 1. We evaluate the success of our approach by measuring the deviation of the detected requirements from the impacted ones.

3. IDEA AND APPROACH OVERVIEW

Among the various software artifacts that are created during the development of a software system, the source code is *the* artifact that is changed whenever a change in the software system behavior is needed. This is because no change in the system behavior happens if the code is not changed. Implementing a code change requires an impact analysis that is carried out at the code level to identify all the parts that need to be modified. Our idea is to build an approach that takes advantage of the impact analysis that is carried out at the code level to automatically identify the impacted parts in the requirements.

Our approach is automated in the sense that when a maintainer commits a set of changes to the source code, he or she can press a button to receive a ranked list of the requirements that are probably impacted by this change. When our method achieves a recall of *n*%, the maintainer can then identify *n*% of the truly impacted requirement by examining only the requirements in this list. The activity diagram in Figure 2 shows an overview of the maintenance process when using our approach. After implementing changes in the code (A1) and committing these changes to a version control system (A2), the changes are automatically analyzed to detect whether they impact requirements or not (A3). If no requirements-impacting changes are detected, nothing is displayed to the maintainer (E1). If, however, requirements-impacting changes are detected, then these changes are traced to the requirements specification (A4), and the related requirements are displayed to the maintainer in a ranked list (A5). The maintainer will then go through this list, identify the truly impacted requirements, update them accordingly, and discard the rest (A6). When comparing our approach to the alternative of manually analyzing the full requirements specification, our approach saves a big amount of effort, thus encouraging maintainers to update the requirements right after modifying the code. As our approach will not achieve 100% recall in most cases, the maintainer will miss some impacted requirements when using our approach. However, a fully manual impact analysis of the whole requirements specification would most probably also miss some impacted requirements because of human misclassification. Hence, as long as our approach achieves high recall (which it actually does in the projects we evaluated; cf. Section 7), missing a few impacted requirements is acceptable.

From the implementation perspective, our approach is composed of three steps: (1) identifying the relevant changes in the commit (*differencing step*); (2) identifying the requirements that are impacted by the changes (*tracing step*); and (3) displaying the impacted requirements to the user (*displaying step*).

Differencing step The goal of the differencing step is to detect whether or not the commit includes changes that impact the requirements. The challenge in this part is to find an *automated* way to detect those code changes that are impacting requirements. To address this challenge, we conducted an exploratory study where we explored the relations between changes in code and changes in the external

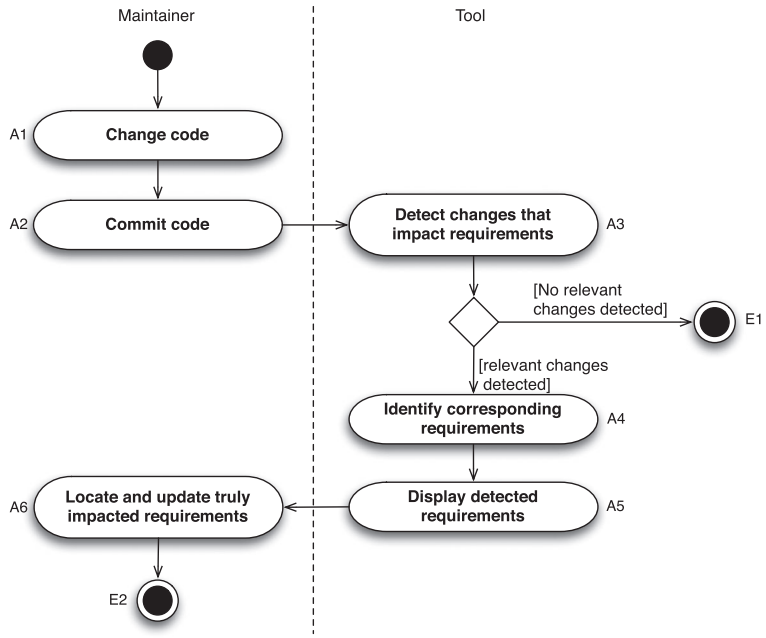

Figure 2. Activity diagram of the maintenance process using our approach.

behavior of an open source software system (Section 4.1). Based on the study, we came up with a set of heuristics about the effect of code changes on the external behavior. We then used these heuristics to develop a differencing approach that detects only code changes that are likely to impact requirements. More details about the exploratory study and the concrete implementation of the differencing part are presented in Section 4.

Tracing step The goal of this step is to trace the relevant changes that were identified in the previous step to the requirements specification in order to identify the requirements that are likely to be impacted. The challenge in this step is to find a tracing technique that is *effective*. The tracing technique we propose is composed of two parts. The first part aims at gathering relevant keywords about the change and its context by extracting terms from the changed elements in the code. In the second part, the keywords are traced to the requirements, and a list of likely impacted requirements is generated. Each requirement in the list is associated to a value that represents the likelihood of the requirements to be impacted by the change. As mentioned earlier, we do not require any manual traces. The tracing step is detailed in Section 5.

Displaying step The goal of the displaying part is to display the results in a convenient way that motivates the maintainer to update the impacted requirements that are detected. We propose two options for displaying the detected requirements. The first option is to present the requirements in the form of a ranked list, where the requirements at the top are more likely to be impacted than the ones below. The second option is to display the complete requirements specification and use color to highlight the parts that are likely to be impacted. The intensity of the color used for highlighting a requirement reflects the likelihood of that requirement being impacted. Implementing the displaying is a pure engineering problem. Therefore, it is not covered any further in the paper.

4. IDENTIFYING RELEVANT CHANGES

The code should implement the needs and requirements of the stakeholders. Therefore, the source code is changed whenever there is a need to adapt the behavior of the system to new requirements. However, not all changes in source code impact the requirements. In fact, many of the code changes are

refactoring, bug fixes, changes in implementation details, and so on. So our goal is to develop an approach that automatically identifies the code changes that are likely to impact requirements. The first results of this approach have been published in [5]. These results include observations made during an empirical study about the relations between changes in the source code and changes in the external behavior of the system. To make the paper self-contained, we include these observations in Section 4.1.

4.1. Exploratory study of the relations between changes in code and changes in requirements

To find out which type of source code changes are likely to impact the requirements, we conducted an exploratory case study where we looked at relations between changes in source code and changes in requirements. We looked at some simple source code change patterns and their impact on requirements. Examples of changes we considered are changes in method bodies, changes in method signatures, addition/deletion of elements, and changes in private elements.

We consider a change as requirements-impacting if it impacts 'the expected behavior of a system or system component in terms of its reaction to given input stimuli and the functions and data required for processing the stimuli and producing the reaction' [4]. Thus, in the terminology of [4] and [7], we address changes of functional requirements (where this change of expected behavior is the main concern [4]) as well as non-functional requirements that are represented operationally [7] (such as a security requirement that describes the login procedure of the system). These changes impact the external (visible) behavior of the system. Therefore, in the rest of the article, we call all the changes that are likely to impact the external behavior of a software system *relevant changes*, and we call all the changes that do not impact the external behavior, such as refactoring and bug fixes, *irrelevant changes*.

We conducted the study on ZXing,[1] an open source project for a barcode reader that is developed in Java.

The research question of this study is:

RQ1: What heuristics can be used to identify the source code changes that impact the external behavior of the system?

We compared two versions of the source code and made observations about how to differentiate between changes that relate to changes in system behavior and changes that are refactorings or bug fixes. In our exploratory study, we went through all the changes between the versions 1.6 and 1.7 of ZXing manually and studied how requirements-impacting changes differ from refactorings and bug fixes. Then, we used some randomly selected packages, to study the changes in detail and confirm the observations we made. In this section, we present the six observations we made and what heuristics for identifying relevant changes we could derive from them.

Observation 1: changes in method bodies are in most cases related to refactoring and/or bug fixes
Changes in method bodies are among the most frequent changes that are applied to the source code. However, they are not the most important ones in terms of impacting the external behavior of the system. In fact, most of the changes observed in the bodies of methods in the explored project were minor changes that are either refactoring or bug fixes. We also noticed that the few changes in method bodies that relate to additions or extensions of features in the system came along with additions of new elements (e.g. new classes, methods, or fields). For example, in the packages that we chose for a detailed exploration, we identified 33 changes in method bodies. Twenty-three of these changes were due to refactorings and bug fixes (the majority, 19, being refactorings), and six changes were related to the additions of new features. For the other four changes, we could not find what was the intent of the developer behind it. All the changes related to feature extension came along with additions of new elements in the code. Based on these observations, we derive the following heuristic: ignore the changes in method bodies.

[1] http://code.google.com/p/zxing/

Observation 2: additions of new elements (classes, methods, package, and fields) are usually related to the addition or extension of features We noticed that extension and addition of features are in most cases implemented through an addition of new elements in the code, where the names of these added elements usually reflect the implemented feature. It is important to note that there were some cases where the added element only relates to some implementation details. Therefore, it is wrong to assume that *all* additions are extensions. However, we still can derive the following heuristic: additions of new elements (additions of packages, classes, methods, and/or fields) are likely to impact the external behavior of the system.

Observation 3: additions and removals of elements having similar names are usually rename operations When using normal differencing tools, renames are detected as an addition and a removal of two different elements. This can be very misleading as addition of elements is likely to relate to feature extension while renames are simple refactorings. When exploring the ZXing project, we noticed that in many cases, the new name is very similar to the old one (e.g., the field PDF417 was renamed to PDF_417). Therefore, renames could be identified by computing the similarity between the name of the deleted and the name of the added elements.

Observation 4: changes in methods signature are usually related to refactoring Changes in methods signatures (other than renaming the method) were among the frequent changes that we observed when exploring the ZXing project and were in most cases related to refactoring. These changes can affect the visibility of the method (public, private, etc.), its return type (e.g., int, boolean, or object), and/or its parameters (e.g., the type of the parameters). In the packages we used for confirming the observation, all of the signature changes were due to refactoring. The heuristic we derive based on this observation is: ignore signature changes.

Observation 5: changes in private elements can impact the external behavior of the system When starting our exploratory study, we were expecting to find that changes in public elements are likely to impact the external behavior of the system, while private elements will only relate to implementation details. However, this was not the case in the ZXing project, as there were many changes in private elements that impacted the external behavior of the system. Therefore, we include changes in private elements when looking for changes impacting the system behavior.

Observation 6: additions of several methods having the same name are usually related to the same feature In many cases, we noticed that several methods having exactly the same name but different parameters were added to a class. In almost all cases, these methods related to the same feature and had similar behavior. Therefore, we derive the heuristic to consider and analyze only one of the added methods instead of considering them all.

4.2. Approach for detecting relevant code changes

In this section, we present our approach for identifying the changes in the code that are likely to impact the external behavior of the system and thus requirements. We built our approach based on the heuristics presented in Section 4.1, about the relations between changes in code and changes in the system external behavior. As the heuristics are based on observations made on a project written in an object-oriented programming language, the approach should work on similar project types. In the rest of this article, we assume that the source code on which we apply our approach is composed of the following elements: packages, classes, methods, and fields.

Our algorithm for detecting relevant changes is composed of two parts: (1) the comparing part, where we compare all the elements in the code to detect the ones that have been added and removed; and (2) the filtering part, where we filter out the additions and removals that are due to renames.

In the comparing part, the main two changes our approach aims at detecting are (1) the addition and (2) the deletion of elements in the code, where an element can be a package, a class, a method, or a field. When focusing on addition and removal only, we are likely to identify the changes related to feature extension, addition, and deletion (Observation 2) while ignoring refactoring and bug fixes that show up as changes in method bodies (Observation 1) and in elements signature (Observation 4).

The comparison is carried out as follows. First, we compare the packages in both versions and detect those that have been added or removed. The comparison is based on the name only; therefore, a package is considered as added to the new version (respectively removed from the old version) if

there is no other package having the same name in the old version (respectively the new version). Second, we go through each of the packages that exist in both versions, and for each package, we look for the classes that have been added or removed. Third, we go through each of the classes that exist in both versions and look for the methods and fields that have been added or removed. Both public and private elements are considered in the comparison as they both are likely to impact the external behavior of the system (Observation 5).

As our comparison is carried out based on element names, renames are detected as a simultaneous addition and removal of two elements. Therefore, we try to identify renames and filter them out. For this, we consider the similarity between the names of the added and the removed elements as well as the call hierarchy of elements. If the added and deleted elements belong to the same parent element (e.g., two fields belong to the same class) and if they have similar names, then the change is considered as a rename (Observation 3) and is filtered out by our approach. There are several ways to compute the similarity between two strings of characters. One of the popular measures is the Levenshtein distance [8], which is calculated as the minimum number of edits needed to transform one string into the other. The edits considered for the Levenshtein distance are insertion, deletion, or substitution of a single character. In the case of methods and classes, we also explore the call hierarchy of the elements: if the added and the deleted elements have the same call hierarchy, then there is a rename.

The output of this step is a set of source code changes that are composed of additions and deletions of elements and that are likely to impact the external behavior of the system and thus its requirements. These changes are then traced to the requirements specification as detailed in the next section. If no relevant changes are detected, then the commit is considered not to be requirements-impacting and is thus ignored.

5. TRACING CHANGES TO THE REQUIREMENTS

In this section, we discuss the tracing strategy for identifying impacted requirements. The tracing approach is composed of two steps. In the first step, we prepare the data to be traced to the requirements (Section 5.1). This data is composed of keywords extracted from the code to describe the change. The keywords are grouped by class. In the second step, we trace the extracted keywords to the requirements using a tracing approach that is based on information retrieval (IR), and we generate a ranked list of the requirements that are likely to be impacted (Section 5.2).

5.1. Keywords extraction

The goal of this step is to extract as much relevant information as possible about the change in order to trace it to the requirements. The tracing is carried out based on the textual similarity between the information describing the change and the requirements. Therefore, it is important to gather as many relevant keywords as possible about the change and its context, but at the same time, the keywords should also be specific enough to the change in order to increase the precision of the tracing. In our approach for extracting keywords, we consider three sources of information: (1) the names of the elements impacted by the change; (2) the call hierarchy of the changed elements; and (3) the documentation of the changed elements. We detail how we extract the keywords from each of these sources in the remainder of this section.

Using meaningful names when coding is one of the most important coding practices. Therefore, in many projects, the name of an element reflects its intended behavior (for example, in a library management system, the method for managing book borrows is likely to be called *borrowBook*). We consider names of the changed (added or removed) elements as a valuable source of information about the change itself. If the changed element is a class, we will consider its name and the names of the methods and fields it contains. If the changed element is a method or a field, then we consider its name as well as the name of its parent class. The reason is that parent and sub-elements usually give information about the context of the change. As the element names are likely to contain several keywords, we split these names into keywords based on the naming convention used (e.g., the

Table I. Elements used for extracting keywords for each type of change.

Changed element	Names	Documentation	Call hierarchy
Package	Package, sub-classes	Package	No
Class	Class, sub-methods, sub-fields	Class	Yes
Method	Method, parent class	Method, parent class	Yes
Field	Field, parent class	Parent class	No

camelCase convention). Then, we include these keywords in the list of terms to be traced to the requirements. Column 2 of Table I presents the element names that we include in the list for each type of changed element.

The documentation of classes and methods are likely to include a description of the behavior and purpose of the element in natural language. Therefore, we also include the keywords contained in the documentation of change-related elements to the list of terms to trace (column 3 of Table I).

To get the context of the change, we consider not only the changed elements, but also their call hierarchy. By call hierarchy, we mean all the methods/classes that are invoking the changed method or those invoking the constructor of the changed class. The invocation can be either direct or via other methods/classes. Details about when the call hierarchy is used are given in column 4 of Table I. Elements from external packages are not in the call hierarchy, and the depth of the call hierarchy can be set by the user.

The choice for the keywords that are included is based on our experience and intuition regarding what elements are likely to include relevant information about the change. There are other possibilities regarding the choice of keyword to be included, whose effect could be explored in future work.

We group the terms related to the change by class. The reason behind this is that considering each single change separately is very fine grained, as a change in a single element might not be relevant by itself. On the other hand, elements contained in one class usually relate to the same concept; therefore, grouping changes by classes results in more keywords relating to the changes without losing the specificity of these keywords to the changes. Finally, we filter out stop words and common words in the project in order to reduce the number of irrelevant terms in the list.

5.2. Tracing

After doing the keyword extraction, we obtain several lists of keywords, where each list contains the terms related to the changes in one class. We trace these keywords to the requirements specification using an IR-based tracing technique. The advantage of IR-based tracing is that it is fully automated. There are several IR-based tracing techniques and tools available and can be used, such as Retro [9], which we used for the evaluation. The results of the IR-based tracing is a ranked list of requirements for each changed class. As there are several changed classes, we get a separate ranked list for each class. We use these lists to compute a final list that indicates the relevant requirements to the maintainer. We compute the final list in the following way: we give a score to each requirement appearing in the lists generated by the IR-based tool according to their rank. Let us assume that we are tracing to a requirements specification that includes 300 requirements. Then, for each list, the top requirement gets the score 300, then the second requirement gets the score 299, and so on. Afterwards, we sum up, for each of the requirements, all the scores from the different lists to get the final score of the requirement. Then, we sort the requirements according to their final scores. With this method, the rank of a requirement depends both on its rank in the initial lists and on how often it appears in the lists. This will allow filtering out the ranks obtained from tracing a changed class that is either irrelevant or very generic.

6. TOOLS

Applying our approach is meaningful only when it is supported by tools. Therefore, we developed a prototype that automatically runs the different steps of the approach. The prototype is composed of

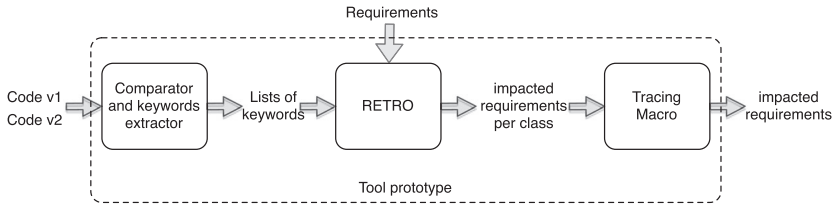

Figure 3. Prototype tool set implementing our approach.

three tools, which are presented in Figure 3. The *comparator* detects the relevant changes in the code and extracts keywords describing the changes. *Retro* traces the keywords to the requirements and generates a ranked list of likely impacted requirements for each class. The *tracing macro* combines the results obtained from Retro and generates the final list of likely impacted requirements.

Comparator The comparator tool compares two versions of source code, detects relevant changes, and extracts keywords describing each of the changes. The tool is based on an existing Java library to compare Java APIs called JDiff [10]. JDiff has several similarities with our comparing technique, as it detects addition and removal of elements in the code and ignores changes in method bodies. We adapted JDiff to our approach by making it ignore changes that are not relevant in our case, such as changes in methods signature. We configured the tool so that it detects changes in private elements. We also implemented a comparator that compares the names of added and removed elements and their call hierarchy in order to detect renames. Name comparing is based on the Levenshtein distance. For the experiments we run with the tool, renames are detected if they have a Levenshtein distance that is equal or less than 2 or if they have the same call hierarchy and a Levenshtein distance that is equal or less than 5. The tool automatically extracts keywords from changed classes as presented in Section 5.1 and generates a list of textual files, where each file contains the keywords related to relevant code changes in one class. In the current implementation of the tool, extracting keywords from package documentation is not supported yet. However, this has no effects on the results of the evaluation we made, as in all projects we used, there was no addition or removal of packages. For the experiment, we set the call hierarchy depth to 2.

Retro For the tracing, we used an existing tool called REquirements TRacing On target (Retro) [9]. Retro, which is an IR-based tool, takes as input two lists of textual documents and returns a list of candidate links that are ranked based on the similarity between the documents. In our case, we give Retro the list of files, where each file contains the keywords related to a changed class, which we obtained from the comparator, and a list of files containing the requirements (one requirement per file) as input. We used Retro with the default configuration, which is the *vector space retrieval* with *term frequency–inverse document frequency* (*tf-idf*) term weighting. Retro includes other functionalities, which we did not use in our evaluation, such as entering analyst feedback to improve the tracing. As output, we obtain a list of requirements for each changed class. The requirements are ranked according to their similarity to the keywords. Retro also assigns values representing the similarity, but we do not use these values in the current version of our approach.

Tracing macro To obtain the final list of likely impacted requirements, we merge the lists obtained from Retro using a macro that implements the scoring technique presented in Section 5.2. The output of the macro is one list of requirements that are ranked according to their likelihood of being impacted by the change. The comparator tool, the Retro tool, and the macro can be easily integrated into one tool if needed.

7. EVALUATION

7.1. Goal and metrics

To evaluate our approach, we applied it to three case studies: AquaLush, a software project for managing an irrigation system; iTrust, a tool for managing medical data; and Connect, an open

source project for managing and exchanging data of patients between health organizations in the US. For our evaluation, we use the goal-question-metric method. We define our goal according to the template developed by Basili in [11]. The goal of the evaluation is to

> Analyze *our approach for identifying impacted requirements* for the purpose of *evaluation* with respect to *the correctness of the output* from the point of view of *maintainers* in the context of *the co-evolution of textual requirements specification and object-oriented source code.*

To assess the goal, we define the following two questions.

> Q1: How good is the approach in identifying the commits that impact requirements and in filtering out those that do not impact requirements?

This question relates to the first part of our approach, where we detect whether a commit is likely to impact requirements or not based on the heuristics presented in Section 4.1. This is useful information in the context of requirements update as it informs the maintainer whether the code changes are likely to introduce inconsistencies with the current requirements specification or not. Successfully filtering out the commits that do not impact requirements is also important as it reduces the number of irrelevant interruptions for maintainers and thus increases their trust in the approach.

> Q2: For relevant code changes, that is, those that impact requirements, how good is the approach in identifying the requirements that are impacted?

Our approach allows identifying the impacted requirements statements in the requirements specification so that the maintainer does not need to look for them manually. In this question, we evaluate how good the approach is in identifying as many impacted requirements as possible without introducing many requirements that are not impacted by the change.

Metrics for question 1 As Q1 is about classifying whether a commit is requirements-impacting or not, we use three measures from the classification context, which are the *true positive rate* (also called *sensitivity* or *recall*), the *true negative rate* (also called *specificity*) and the *overall accuracy* ([12] p. 138). The true positive rate (*TPR*) evaluates how good the approach is in identifying as many relevant commits as possible, and the true negative rate (*TNR*) evaluates how good the approach is in ignoring as many irrelevant commits as possible. The overall accuracy (*A*) evaluates the ability of the approach to identify as many relevant commits as possible and to ignore as many irrelevant commits as possible.These measures are calculated based on a coincidence matrix (Table II) that includes the number of relevant commits that are detected (true positives = *TP*), the number of irrelevant commits that are ignored (true negatives = *TN*), the number of irrelevant commits that are detected (false positives = *FP*), and the number of relevant commits that are ignored (false negatives = *FN*). More formally, the measures are defined as follows:

Table II. A simple coincidence matrix [12].

		True Class: Positive	True Class: Negative
Predicted Class	Positive	True Positive Count (TP)	False Positive Count (FP)
Predicted Class	Negative	False Negative Count (FN)	True Negative Count (TN)

M1: Accuracy (A) is obtained by dividing the total correctly classified positives and negatives by the total number of samples:

$$A = \frac{TP + TN}{TP + TN + FP + FN}$$

M2: True positive rate (TPR) is obtained by dividing the correctly classified positives by the total positive count:

$$TPR = \frac{TP}{TP + FN}$$

M3: True negative rate (TNR) is obtained by dividing the correctly classified negatives by the total negatively classified count:

$$TNR = \frac{TN}{TN + FP}$$

In the ideal case, where the approach returns true positives and true negatives only, we get $A = 1$, $TPR = 1$, and $TNR = 1$, and in the worst case, where only false negatives and false positives are returned, we get $A = 0$, $TPR = 0$, and $TNR = 0$. Therefore, we aim at obtaining values of A, TPR, and TNR that are as high as possible.

Metrics for question 2 As question Q2 is about identifying a certain class of requirements (the impacted ones) in a large set of requirements, we use the metrics *recall* (R), *precision* (P), and *fall-out* (F) from the IR field to assess how good the results are [13]. *Recall* (which is similar to the TPR for Q1) evaluates how good our approach is in identifying as many impacted requirements as possible. The higher the recall, the less risk to miss an impacted requirement when updating only those requirements that our approach detects. Thus, recall must be high in order to make our approach effective and practically useful. *Precision* evaluates how good our approach is in identifying impacted requirements only. Better precision makes our approach more convenient to use for the maintainer: the higher the precision, the less false positives he or she has to discard manually from the list of detected requirements. *Fall-out* (which is equivalent to 1-TNR for Q1) evaluates how many irrelevant (i.e., non-impacted) requirements are wrongly detected. The lower the fall-out, the better our approach filters irrelevant requirements, thus yielding a smaller list of potentially impacted requirements. In the ideal case, where the approach returns the impacted requirements only, we get $R = 1$, $P = 1$, and $F = 0$. In the worst case, where only requirements that are not impacted are returned, we get $R = 0$, $P = 0$, and $F = 1$.

Actually, recall is the most important metric in our case. If recall is not high enough, the maintainer cannot rely on our approach as it would miss too many impacted requirements. Low precision is inconvenient as it decreases the efficiency of using the results of our approach, but it does not affect its effectiveness. Also, if our approach provides an effective filter (i.e., high recall and low fall-out), it is efficient even when precision is low: manually finding the truly impacted requirements in the result list produced by our approach requires much less effort than performing a manual impact analysis of the full requirements specification. Fall-out indicates how good our approach filters non-impacted requirements. When limiting the size of the result list by some cut-off rank (see subsequent text), fall-out is inherently limited by the chosen cut-off rank. Nevertheless, fall-out remains to be an indicator for the quality of our filter.

As it is easier for the maintainer to find the impacted requirements when they are suggested early in the ranking, we use recall R_n, precision P_n, and fall-out F_n at cut-off rank n, that is, we only consider the requirements identified with a rank that is lower than n (cf. [13], Section 4.9.3). In other terms, we

evaluate how good the approach is in identifying the impacted requirements within a list that contains n requirements only.

To define the new metrics, we use the same terminology used in the classification context (TP, TN, FP, and FN). The reason is that IR is also a kind of classification, as the elements are classified as relevant (which should be retrieved) and irrelevant (which should be ignored). The coincidence matrix for Q2 includes, for each commit i, the set of impacted requirements that are detected with a rank lower than n by the approach (TPin), the set of requirements that are not impacted and are detected with a rank lower than n (FPin), the set of requirements that are not impacted and are not detected with a rank lower than n (TNin), and the set of impacted requirements that are not detected with a rank lower than n (FNin).

More formally, the measures are defined as follows:

M4: R_n is the average recall for all commits

$$R_n = \frac{\sum_{i=1}^{k} r_{in}}{k}$$

where k is the number of commits considered and r_{in} (the recall for commit i at cut-off rank n) is the proportion of identified requirements out of the impacted ones

$$r_{in} = \frac{TP_{in}}{TP_{in} + FN_{in}}$$

M5: P_n the average precision for all commits

$$P_n = \frac{\sum_{i=1}^{k} p_{in}}{k}$$

where k is the number of commits considered and p_{in} (the precision for commit i at cut-off rank n) is the proportion of impacted requirements out of those identified

$$p_{in} = \frac{TP_{in}}{TP_{in} + FP_{in}}$$

M6: F_n the average fall-out for all commits

$$F_n = \frac{\sum_{i=1}^{k} f_{in}}{k}$$

where k is the number of commits considered and f_{in} (the fall-out for commit i at cut-off rank n) is the proportion of non-impacted requirements that are identified out of the ones that are not impacted:

$$f_{in} = \frac{FP_{in}}{TN_{in} + FPin}$$

To study how early relevant links are suggested by the approach, we increase the cut-off rank n, which starts from 1 and create a precision-recall graph as well as a fall-out-recall graph.

7.2. *Case studies and experimental setup*

To do the evaluation, we used three case studies[2]: AquaLush, iTrust, and Connect. For each of the case studies, we manually identified the ground truth (the relevant commits and the impacted requirements for each commit). Then, we run the tool presented in Section 6, and we evaluated the metrics we defined previously. In this section, we present each of the case studies we used, as well as how we built the ground truth for it.

[2]The requirements specifications used in the three case studies are available at http://www.ifi.uzh.ch/rerg/research/requpdate/experimentdata

Table III. List of changes applied to AquaLush.

Change 1	Allow setting the water allocation for each of the zones separately
Change 2	Add a maximum moisture level: the irrigation should start when the moisture level is lower than the critical moisture level and should stop as soon as the maximal moisture level is reached. The default max level should be 50.
Change 3	Create a log that includes the timestamp for each of the following events: change irrigation mode, setting water allocation, setting irrigation time, setting critical, or maximum moisture level. A button *show log* allows the user the access the log. There should be two buttons that allow the user to browse the log up and down.
Change 4 (bug)	When setting the *times real time* to the maximum (1000), there is a problem in the screen for *control irrigation* (the screen is blinking).
Change 5 (bug)	The sim storage device is not working as it does not store any data.
Change 6 (bug)	In class *Zone*, the method *setIsFailed*, does not set the valve to *closed* after setting it to *failed*. This might lead to the problem that a device is set as failed and open at the same time.
Change 7 (bug)	If there is already less than 1 h before the next irrigation time, the *jump* button should have no effect. However, in the current implementation, jump sets the time to the next irrigation day.
Change 8 (bug)	It is impossible to set water allocation to 0: when setting the water allocation to 0 and clicking on *Accept new settings* then going back to set water allocation, we find that the old value of allocation is restored.

7.2.1. Case study 1: AquaLush

Description AquaLush is a software project for managing an irrigation system that has been originally developed as an illustrative example for a book about software design [14] and has later been extended and used as a benchmark for traceability [15]. AquaLush has a structured requirements specification that is written in natural language. The source code of AquaLush is written in Java with around 11KLOC. The requirements specification of AquaLush, its source code, and all other AquaLush artifacts can be found at the webpage of the AquaLush benchmark[3]. To perform the tracing part of the experiment, we had to delete the section titles as well as the figures in the requirements specification, and we included each requirement statement in a separate textual file. This is because Retro only accepts a flat list of textual files as input for the tracing. The total number of statements considered in the experiment is 337, which is also the number of input requirements used in this experiment.As there is only one release of AquaLush available, we had to develop a second release to run our experiment. Therefore, we prepared a list of eight changes, and we asked a developer to implement these changes. The changes include three new features and five bug fixes. The list of changes is presented in Table III. In order to limit the threats to validity when developing the new version, we asked an external developer to implement the changes. The developer was not involved in our work and did not know why we were developing the new AquaLush release. In order to avoid the risk of having all the changes in one commit, we asked the developer to commit the code after implementing each of the changes and to mention in the commit message what change he had implemented. The developer did an additional commit to fix a bug that was introduced when implementing one of the features. Therefore, we had nine commits in total.

Ground truth The ground truth is composed of two components, which are (1) the set of commits that impact requirements and (2) the set of impacted requirements for each requirements-impacting commit. Deciding which commits are requirements-impacting is straightforward. In fact, we know already which of the implemented changes impact requirements, and we know what change is implemented by each commit. In total, we have six commits that do not impact requirements (commits for changes 4, 5, 6, 7, 8, and the additional bug fix made by the developer), and three commits that impact requirements (commits for changes 1, 2, and 3).To identify impacted requirements, we went manually through the requirements specification of AquaLush and identified what requirements are impacted by changes 1, 2, and 3. We identified eight impacted requirements for change 1, six impacted requirements for change 2, and three impacted requirements for change 3.

[3]http://www.ifi.uzh.ch/rerg/research/aqualush.html

7.2.2. Case study 2: iTrust

Description The second case study we used for the evaluation is the iTrust Medical care project [16], which is a tool for managing medical data that is developed using a combination of Java code and Java Server Pages. The Java code is about 25KLOC. The tool was developed for teaching purposes at the North Carolina University, and it has several code releases and a wiki-based requirements specification that includes functional requirements, non-functional requirements, a glossary, a set of global constraints, and a section for specifying the data format for the input fields. Every semester, students apply new changes to iTrust, and a new source code version is released at the end of the semester. In our experiment, we used the functional requirements of iTrust, which are specified in the form of fine-grained uses cases (Figure 4). In total, there are 39 use cases, which is also the number of input requirements we use in this experiment. For the code, we only considered the part written in Java as our prototype only works on Java code. We used versions 10 (release date: 18 August 2010) and 11 (release date: 7 January 2011) of the source code. To obtain the requirements that correspond as much as possible to each of these releases, we choose a wiki version from a date that is after the code release and before the beginning of the following semester. The reason is that, after the release, the project owners do a cleanup and maintenance for the requirements based on the work carried out by the students. Therefore, we consider the requirements specification as of 3 September 2010 (the 'old requirements') for the source code version 10 (the 'old code version'), and the requirements as of 7 February 2011 (the 'new requirements') for the source code version 11 (the 'new code version') [17].

Ground truth The challenge we had in building the ground truth for iTrust was that the commits were not available. Therefore, we had to manually compare the two versions of source code and classify all the changes according to whether they impact requirements or not. Then, we grouped the changes that relate to the same conceptual change, that is, the same feature, and considered them as one commit. We found 14 conceptual changes, which are presented in the second column of Table IV. For the changes that did not impact requirements, we could not group them by commit as it was impossible to find how they were distributed. Consequently, we could not estimate the count of FP and TN, which are needed to calculate the accuracy *A* and the TN rate *TNR* for Q1. Therefore, we added new metrics, which are $\tilde{A}$, $\widetilde{TPR}$, and $\widetilde{TNR}$ for detecting code classes that include requirements-impacting changes. The formulas for $\tilde{A}$, $\widetilde{TPR}$, and $\widetilde{TNR}$ are similar to those for *A*, *TPR,* and *TNR*, with the difference that

- $\widetilde{TP}$ is the number of detected classes that contain relevant changes;
- $\widetilde{TN}$ is the number of classes containing irrelevant changes that are ignored;
- $\widetilde{FP}$ is the number of classes containing irrelevant changes that are detected; and
- $\widetilde{FN}$ is the number of classes containing relevant changes that are ignored.

UC1 Create and Disable Patients Use Case

1.1 Preconditions: The iTrust HCP has authenticated himself or herself in the iTrust Medical Records system [UC3].
1.2 Main Flow: An HCP creates patients [S1] and disables patients [S2]. The create/disable patients and HCP transaction is logged [UC5].
1.3 Sub-flows:

[S1] The HCP enters a patient as a new user of iTrust Medical Records system. Only the name and email are is provided. An email with the patient's assigned MID and a secret key (the initial password) is personally provided to the user, with which the user can reset his/her password. The HCP can edit the patient according to data format 6.4 [E1] with all initial values (except patient MID) (...)

[S2] The HCP provides the MID of a patient for whom he/she wants to disable [E2]. The HCP provides a deceased date (data format 6.4). An optional diagnosis code is entered as the cause of death.

1.4 Alternative Flows:

[E1] The system prompts the enterer/editor to correct the format of a required data field because the input of that data field does not match that specified in data format 6.4 for patients.

[E2] (...)

Figure 4. Example use case from iTrust requirements specification, version of 3 September 2010 [17].

Table IV. Identified changes and impacted use cases in iTrust.

Conceptual change	Change description	impacted use cases
Change 1	Activity feed	none (new requirement)
Change 2	Enabling appointment editing	UC22
Change 3	Uploading photo	UC4
Change 4	Reason code	UC15, UC37
Change 5	Weight/height charting	UC10
Change 6	Login (added captcha and attempts)	UC3
Change 7	Office visit form (added orc and comment)	UC11
Change 8	Remote monitoring (added height, weight, etc.)	UC34
Change 9	Remote monitoring (get patient data by type)	UC34
Change 10	Display patient's monitoring HCP	UC34
Change 11	Cause of death validation	UC1
Change 12	Notes (format change)	UC11
Change 13	Logging (added logs)	UC5
Change 14	Color options	none (new requirement)

There were 91 classes that changed in total, among which we found 31 that contain requirements-related changes.

The change we made for evaluating Q1 does not affect the metrics used for Q2. To identify the impacted requirements for each of the conceptual changes, we used the same ground truth that we built for a previous experiment in [5], and where we went manually through the use cases and identified the ones that are impacted. Although the current experiment is different from the one performed in [5], as now we trace all the classes related to a conceptual change together, the ground truth is the same for both experiments. The impacted requirements for each of the conceptual changes are reported in the third column of Table IV.

7.2.3. Case study 3: Connect

Description Connect is a an open source project that aims at facilitating the secure exchange of health data between health institutions. The Connect project includes a large amount of documentation such as architecture documents, design documents, interfaces, manuals for use, health information references, issue tracking, release notes, requirements for each release, requirements traceability matrices, and change requests. The source code, which is mainly written in Java, is also publicly available with all the historical changes. In the experiment, we used the Java code that relates to the main product, and which is available under the folder product/production in the repository. The source we used is about 280 KLOC. To run the experiment, we used the requirements traceability matrix (RTM) as well as the Jira tickets referenced in the release note of the release 3.3. The RTM relates several requirements or change requests that were implemented in a release to the corresponding issues in Jira. Having the link between the requirement and the corresponding issue number facilitates the search of the code commits that implement the change.

Ground truth We run the experiment using the changes applied to the Connect software for the release 3.3. We obtained the list of changes from an analysis of the release note,[4] Jira, which is the issue tracking system used for the project, and a document that is called RTM,[5] which actually contains a list of the requirements that were implemented in release 3.3 as well as a link from each of these requirements to the corresponding issue description in Jira. In the RTM, there are 56 requirements. Three of these requirements did not have a link to Jira and thus could not be traced to the commits in the repository. Therefore, we did not include them in the experiment. The release note includes 80 issues. We went through all the issues in the RTM and in the release note and looked for code commits that link to them and that impact Java source code. We found commits that impact source

[4]https://connectopensource.atlassian.net/wiki/display/CONNECTR33/Release+Notes
[5]https://connectopensource.atlassian.net/wiki/download/attachments/9044001/DM_183_Rel3.3_CONNECT_RTM_Doc.pdf

code for 27 of these issues. We classified the commits that correspond to the 27 issues as follows. If the issue is classified as a bug in Jira, then all commits that relate to it are considered as not impacting requirements and thus should not be detected by our tool. If the issue appears as a requirement in the RTM and is also classified as a task in Jira, then the commit that implements it is considered as requirements-impacting and thus should be detected by our approach. There are six issues that are classified as bugs in Jira and also do appear in the RTM. These six bug issues, which were probably added to the RTM because they are considered as major bug fixes, are included in the first category, that is, the corresponding commits are considered as not impacting requirements. For some issues that appear in the RTM and are classified as tasks, there were more than one commit referencing them, which means that the change task was split into sub-changes that were implemented and committed separately. We consider that the first commit is the one that is likely to include the main changes, and thus is the one to be detected as requirements-impacting. As it is not possible for us to decide whether the follow-up commits impact requirements or not, we only consider the first commit in the experiment. We found two commits that relate to several issues, where some issues are classified as bug fixes, and other issues are classified as tasks, but do not appear in the requirements list. We decided to ignore these commits because we did not know whether our tool detects them because of changes that derive from the bug fix or because of changes that derive from the tasks. We ended up with a list of 26 issues and 32 commits. The used commits, the related issues, and their classification are reported in columns 2–4 of Table V.

For the tracing part of the experiment, we trace the code changes to the 53 requirements in the RTM. For each commit, we consider that the impacted requirements are the requirements that are linked to the commit via the Jira issues.

From the version control system, we could obtain the source code for the different commits, but not the compiled version of the code. This impacts our tool, which uses the compiled version to analyze the call hierarchy. Therefore, in this experiment, we run the tool without the call hierarchy feature.

7.3. Results

In this section, we report the results obtained for each of the case studies.

7.3.1. AquaLush

Detecting relevant commits (Q1) When checking for relevant commits, our approach detected four out of the nine AquaLush commits as relevant. The result for individual commits is reported in Table VI. Three of the commits were actually relevant. There are in total three true positive, five true negative, one false positive, and zero false negative. The accuracy (M1), TPR (M2), and TNR (M3) we obtained are $A = (3+5)/(3+5+1+0) = 88.8\,\%$, $TPR = 3/(3+0) = 100\,\%$, and $TNR = 5/(5+1) = 83.3\,\%$.

Detecting impacted requirements (Q2) To answer Q2, we evaluated the recall (M4), precision (M5), and fall-out (M6) for various cut ranks n. In Table VII, we report the list of requirements obtained for a cut rank $n = 25$, and we color the impacted requirements. We also color the related requirements, which are requirements that are consistent with the new implementation (not outdated) and are interesting because they are directly related to the change. We highlighted them as we think that they can give the user information about the context of the change. Detecting the related requirements can be interesting as well as it might support the maintainer, but it is not compulsory; therefore, we do not consider it in our metrics. Each requirement has an identifier, which is composed of the term SRS and then the ID number of the requirement as found in the benchmark for traceability. We report the precision/recall values and fall-out/recall values obtained for various cut ranks in Figures 5 and 6, respectively.

The maximum precision value we obtained is $P_{20} = 23.3\,\%$ for a recall of $R_{20} = 73.6\,\%$ and a fall-out of $F_{20} = 4.6\,\%$. This means that by looking at 20 requirements only, the maintainer is able to detect 73% of the impacted requirements after each commit and that only 4.6% of the requirements that are not impacted will be suggested to the maintainer as impacted, while the rest, 95.4% of the requirements that are not impacted, will be filtered out.

Table V. Commits and issues used in the experiment with the Connect project

	Commit ID in Git	Related Issue	Jira classification	Is requirements-impacting (Ground-truth)	Detected as requirements-impacting by our approach
Commits linked to issues in RTM	a712db6e82384072f91d70bc73bdce1ad2736523	GATEWAY-847	task	yes	yes
	ca5a31507b0c4067d8a3321cded75eabf57bd286	GATEWAY-840	task	yes	yes
	ca4ad44a1bd17fc72dd1d5f83e84470492a17c7b	GATEWAY-1359	bug	no	yes
	6d43a29611a7ebb235d54645a0bc5cf71d8fa657	GATEWAY-839	task	yes	yes
	3ee520200c79e8ba3629f0c0acfc1e2a3307bf2f	GATEWAY-841	task	yes	yes
	00287e0453bb285824817d8959f1fa5cc6925202	GATEWAY-843	task	yes	yes
	9a4c9c1e75d24afd9f928cc5bd21ce35a33b0f3b	GATEWAY-321	sub-task	yes	yes
	9ccb971359624e9330e23272558bd0fb5cb7350c	GATEWAY-326	bug	no	no
	43c230c56d375cdff72968e041056649538141f7	GATEWAY-306	bug	no	no
	526f320856c4dfe8f96350449f4c85a3fee7ae31	GATEWAY-1123	sub-task	yes	yes
	8aa6da964c4894201ed08a6187249b9bd64944e7	GATEWAY-1121	task	yes	yes
	d5c24f56ff668a76b57ec57a77956561f1e8cd83	GATEWAY-1241	sub-task	yes	yes
	cfec670354f9a90bcbc01cf8f8b00cd6678afde4	GATEWAY-1242	sub-task	yes	yes
	85b93bc860ecec7eabc73da1803212c68720ad28	GATEWAY-1243	sub-task	yes	yes
	c4f3323c90979baccbe5d965f3ac3bcbe5c57adc	GATEWAY-1244	sub-task	yes	yes
	acd1bb54351444aa0100cffc8720715dfd76bc73	GATEWAY-1346	sub-task	yes	yes
Commits related to issues in release note	eed6e5882c4ac569042dcb2bc3dad4edaa57726e	GATEWAY-532	bug	no	yes
	1565ec7a112ee99ac07bdf7c3b43462b0e408b13	GATEWAY-402	bug	no	no
	53b3b52468182bb4afea6a8b9a98cad57b199368	GATEWAY-400	bug	no	no
	9aaf1eb5edd02b5ba5c5a8ec1bc0b397c4125c81		bug	no	no
	c0026edf59a2da8ef783f704a9f8590fb9ee6532	GATEWAY-394	bug	no	no
	0d0c53a25ecf584dc6ef8e2a6f4e5bdcb4643ca1			no	no
	3ba6b4af1ac6546ed63767f5e548ade15e6b5b46			no	no
	3f0a5f666ec4074a392ae0945fffc70d0b2d6fc9			no	no
	7defbccb45632534c6d4ea41c091ac4d7de4860e	GATEWAY-396	bug	no	no
	e7561ba0a89de11f13a53c98cc22f6f79f80f3e1	GATEWAY-236	bug	no	yes
	9f5ab8cf653477748f4500df1d9c98e5edf31129	GATEWAY-245	bug	no	yes
	0525f493c2a6f5b6466e8626137531efa0396cb6			no	yes
Commits related to issues in both	51a49cc4ea205953d18dbd0a44dd0196e0c937db	GATEWAY-393	bug	no	no
	98db974125c4a668dc4b829530fd2fa735f83741	GATEWAY-302	bug	no	no
	b67b1e0c5b4baae8d89584ffebe8cb5c488a2fe4	GATEWAY-299	bug	no	no
	3a72a13720255dbe56749dd5ae543cfd71ca95bb			no	yes

True positive
True negative
False positive

Table VI. List of changes applied to AquaLush.

Change commit	Requirements-impacting (ground truth)	Detected by our approach as requirements-impacting
Change 1	Yes	Yes
Change 2	Yes	Yes
Change 3	Yes	Yes
Change 4	No	No
Change 5	No	Yes
Change 6	No	No
Change 7	No	No
Change 8	No	No
Additional commit	No	No

Table VII. Identified requirements for AquaLush changes with a ranking below 25.

Rank	Change 1	Change 2	Change 3
1	SRS383	SRS358	SRS237
2	SRS33	SRS24	SRS24
3	SRS384	SRS367	SRS97
4	SRS254	SRS28	SRS53
5	SRS53	SRS366	SRS293
6	SRS30	SRS253	SRS69
7	SRS378	SRS368	SRS252
8	SRS31	SRS369	SRS238
9	SRS52	SRS377	SRS277
10	SRS32	SRS27	SRS52
11	SRS379	SRS97	SRS98
12	SRS145	SRS376	SRS291
13	SRS29	SRS25	SRS48
14	SRS186	SRS360	SRS240
15	SRS24	SRS187	SRS112
16	SRS48	SRS26	SRS32
17	SRS153	SRS374	SRS109
18	SRS69	SRS359	SRS333
19	SRS380	SRS53	SRS110
20	SRS98	SRS69	SRS150
21	SRS47	SRS52	SRS289
22	SRS97	SRS113	SRS27
23	SRS382	SRS167	SRS99
24	SRS49	SRS277	SRS96
25	SRS381	SRS109	SRS269
Missed	0	0	2

Outdated
Related

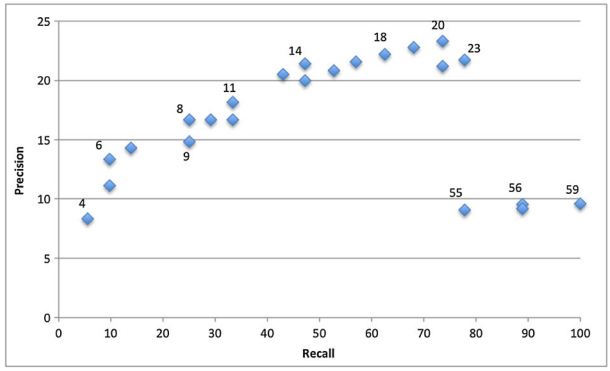

Figure 5. AquaLush: precision/recall at different cut ranks; the respective cut rank *n* is annotated to the data points.

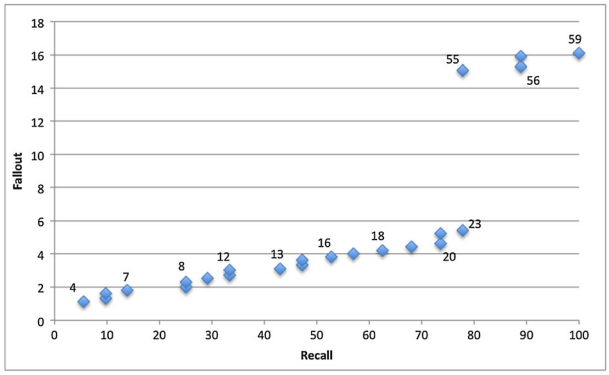

Figure 6. AquaLush: fall-out/recall at different cut ranks; the respective cut rank n is annotated to the data points.

Full recall, $R_{59} = 100\,\%$ is obtained at cut rank $n = 59$ for a precision of $P_{59} = 9.6\,\%$ and a fall-out of $F_{59} = 16\,\%$. This means that the list proposed to the maintainer includes all of the impacted requirements and filters out 84% of those that are not impacted.

7.3.2. iTrust

Detecting relevant commits (Q1) For the iTrust project, 12 out of the 14 conceptual changes were detected as relevant. So we get 12 true positive and two false negative. This gives a recall of $TPR = 12/(12+2) = 85.7\,\%$. When considering classes instead of commits (Section 7.2.2), the approach detected 33 classes among which 26 actually contained requirements-related changes, and it ignored 53 out of the 60 classes containing irrelevant changes. So we get 26 true positives, seven false positives, 53 true negatives, and five false negatives. This results in $\tilde{A} = (26+53)/(26+53+7+5) = 86.8\,\%$, $\widetilde{TPR} = 26/(26+5) = 83.8\,\%$, and $\widetilde{TNR} = 53/(53+7) = 88.3\,\%$.

We did not obtain a recall of 100% because two conceptual changes were missed. The two missed changes in iTrust are the following. (1) A change of the input format for notes, where a hash tag has been added as an allowed character. As this change only was made inside a string constant in the *ValidationFormat* enumeration class, our tool did not detect it. (2) An addition of a new condition in the *PatientValidator* class to validate that no patient can be marked as dead unless the cause of death is specified. As this was implemented by adding an 'if' statement inside the body of a method, it could not be detected by our tool. To summarize, both undetected changes were related to the validation of input forms only.

Detecting impacted requirements (Q2) In this paragraph, we report the recall, precision, and fall-out obtained when tracing the conceptual changes that were detected in the previous step to the requirements specification of iTrust. It is important to mention that we did not consider the changes that only result in addition of new requirements without implying any changes in the existing requirements (changes 1 and 14) because our current approach addresses outdated requirements and not missing ones. The precision/recall values and fall-out/recall values we obtained for different cut ranks are presented in Figure 7 and 8, respectively. We also report the ranks of the impacted use cases as identified by our approach in column 3 of Table VIII.

The maximum precision value we obtained is at cut rank 1, where we have $P_1 = 44.4\,\%$, $R_1 = 44.4\,\%$, and a fall-out of $F_1 = 1.4\,\%$. Full recall, $R_{34} = 100\,\%$ is obtained at cut rank $n = 34$ for a precision of $P_{34} = 3.2\,\%$ and a fall-out of $F_{34} = 86.8\,\%$. At cut rank $n = 7$, the recall is $R_7 = 72.2\,\%$ for a precision of $P_7 = 11.1\,\%$ and a fall-out of $F_7 = 16.4\,\%$. This means that the maintainer can detect more than 70% of the impacted requirements by looking at a list of seven requirements after each commit.

When looking carefully at the ranks reported in Table VIII, we notice that most of the ranks are good (seven or better) except for changes 2, 3, and 4. When investigating the reasons behind the bad ranks,

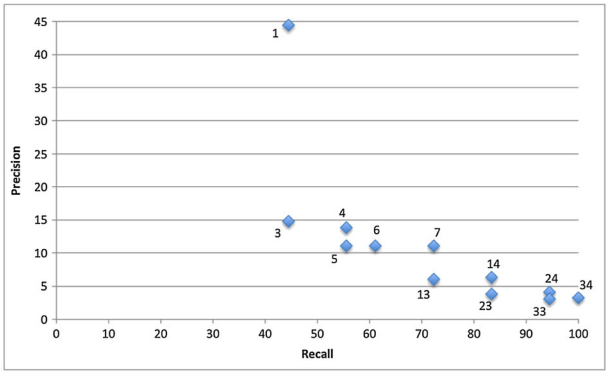

Figure 7. iTrust: precision/recall at different cut ranks; the respective cut rank *n* is annotated to the data points.

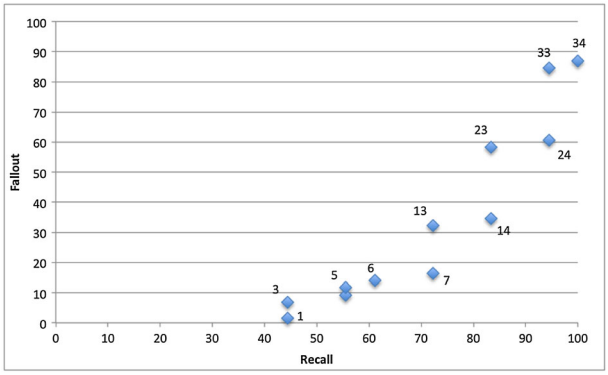

Figure 8. iTrust: fall-out/recall at different cut ranks; the respective cut rank *n* is annotated to the data points.

Table VIII. iTrust rank results.

Change	Impacted use cases	Rank of impacted use case
Change 1	None	New requirement
Change 2	UC22	14
Change 3	UC4	24
Change 4	UC15, UC37	6,34
Change 5	UC10	7
Change 6	UC3	1
Change 7	UC11	4
Change 8	UC34	1
Change 9 and 10	UC34	1
Change 11	UC1	Change not detected
Change 12	UC11	Change not detected
Change 13	UC5	1
Change 14	None	New requirement

we found the following. For change 2, the problem was related to the use of abbreviated terms in the code, namely, the term *appt* was used as abbreviation to *appointment*. The abbreviated terms could not be matched to the complete terms in the requirements specification, and this resulted in the bad rank. Changes 3 and 4 were extensions of existing features, so new terms have been added, which did not appear in the requirements specification. Additionally, for both of these cases, the new elements that have been added to the code were called from the jsp classes only, so there was no call hierarchy available,

and thus we could not gather more information about the context of the change. Therefore, the tracing of these changes to the code was not efficient.

7.3.3. *Connect*

Detecting relevant commits (Q1) Among the 32 Connect commits that we used in the experiment, all of the 13 commits that related to issues marked as tasks or subtasks are detected by our approach (true positives). For the 19 commits that relate to issues marked as bugs, only six are detected (false positives), the other 13 were filtered out (true negatives). As none of the first commits that relate to a task was missed, there are no false negatives. The accuracy (M1), TPR (M2), and TNR (M3) obtained are $A = (13+13)/(13+13+6+0) = 81.2\,\%$, $TPR = 13/(13+0) = 100\,\%$, and $TNR = 13/(13+6) = 68.4\,\%$. When looking closely at the false positives, we found that for one commit, the change was a rename in a package that was detected by our tool as an addition and a removal of packages. For the other changes, the detection was due to addition of new methods or attributes in the code.

Detecting impacted requirements (Q2) We did the tracing for 13 commits that were detected by our tool and which relate to the 13 relevant issues mentioned previously. We report the ranks obtained for the related requirement for each of these commits in Table IX. For eight commits, the corresponding requirement was ranked first. For 11 out of the 13 commits, the corresponding requirement was ranked third or better. There were only two issues for which the source requirement could not be detected. When looking closely, we found that for one of them, the main term in the requirement is 're-identification'. This term was typed as 'reidentification' in the code. Therefore, the tracing tool failed in linking them to each other. This could be fixed by adding a naming convention rule in our tool. For

Table IX. Connect rank results.

Requirement	Rank
GATEWAY-847	2
GATEWAY-840	1
GATEWAY-839	1
GATEWAY-841	2
GATEWAY-843	3
GATEWAY-321	Not identified
GATEWAY-1123	1
GATEWAY-1121	1
GATEWAY-1241	1
GATEWAY-1242	1
GATEWAY-1243	1
GATEWAY-1244	1
GATEWAY-1346	Not identified

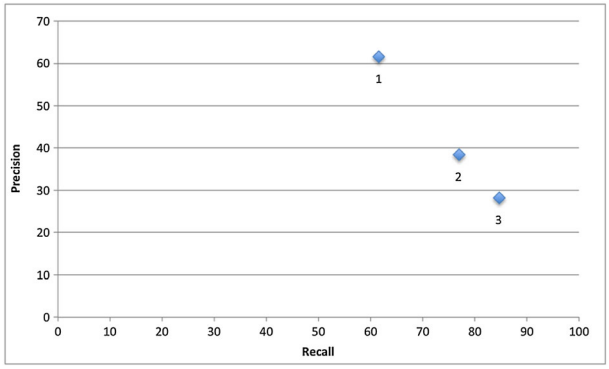

Figure 9. Connect: precision/recall at different cut ranks; the respective cut rank n is annotated to the data points.

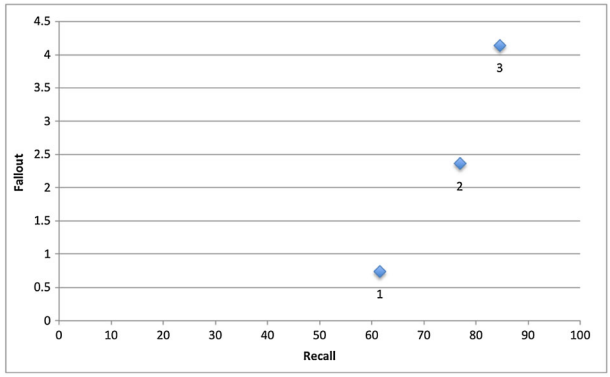

Figure 10. Connect: fall-out/recall at different cut ranks; the respective cut rank n is annotated to the data points.

Table X. Results summary.

		AquaLush		iTrust		Connect	
Detecting relevant commits (Q1)	Accuracy	88.8		86.8		81.2	
	TPR (sensitivity)	100		83.8		100	
	TNR (specificity)	83.3		88.3		68.4	
Detecting outdated requirements (Q2)	*Cut rank*	$n=23$	$n=59$	$n=1$	$n=7$	$n=1$	$n=3$
	Recall	77.7	100	44.4	72	61.5	84.6
	Precision	21.7	9.6	44.4	9.7	61.5	28.2
	Fall-out	5.4	16	1.4	19	0.7	4.14

TPR, true positive rate; *TNR*, true negative rate.

the other one, we found that the issue was not resolved in the first commit, but rather in the last commit, which was committed a few months later. When tracing the last commit, the corresponding requirement could be identified with a rank equal to 4.

The precision/recall values and fall-out/recall values we obtained for different cut ranks are presented in Figures 9 and 10, respectively. The Maximum precision value we obtained is at cut rank 1, where we have $P_1 = 61.5\,\%$, $R_1 = 61.5\,\%$, and a fall-out of $F_1 = 0.7\,\%$. The highest recall value $R_3 = 84.6\,\%$ is obtained at cut rank $n = 3$ for a precision of $P_3 = 28.2\,\%$ and a fall-out of $F_3 = 6\,\%$.

7.4. Summary

The results obtained from the experiments are summarized in Table X. The recall, precision, and fall-out values are given for two cut rank values: a cut rank with good recall and fair precision and a cut rank with a high recall and lower precision. For all case studies, our approach succeeded to detect the relevant commits with an accuracy higher than 86%, a sensitivity (TPR) higher than 80% and a specificity (TNR) that is higher than 68%. For the identification of impacted requirements, a recall higher than 70% could be obtained for all studies while filtering out more than 80% of the requirements that are not impacted and with a precision of about 10% to 30%.

8. DISCUSSION

8.1. Approach evaluation

The results obtained in the evaluation demonstrate that our approach succeeded in achieving the goals of identifying requirements-relevant code commits and identifying the impacted requirements relating to them. In fact, for all projects, the approach detected most or all of the relevant commits and most or all of the requirements related to them while filtering out more than 68% of the irrelevant commits and more than 80% of the irrelevant requirements. The precision

obtained for detecting impacted requirements is around 10% and 30%. Although these precision values are medium, our approach is still likely to save significant effort to the maintainer because of the low fall-out. In fact, obtaining 10% precision for 9.1% fall-out when looking for one outdated requirement out of 100 means the following: to identify the outdated requirement, the maintainer needs to look at 10 requirements only compared with looking at 100 requirements if the update was to be carried out manually. Therefore, the fall-out reflects the effort saving better than the precision in this case.

As the approach is automated, no additional effort is required for using it. The approach gave good results for three case studies that have different characteristics (project size, system type, type, and structure of requirements specification, number of requirements, etc.); therefore, we expect it to perform well for other software systems as well.

The main two limitations of the approach are that (1) it might ignore some of the relevant commits and (2) it might miss some of the impacted requirements. The first problem relates to the compromise between identifying all relevant changes and identifying relevant changes only. If we extend the approach to cover more code change patterns, then the precision of the approach will decrease, and there will be more irrelevant commits that are considered. The second problem is due to the limitation of the IR-based tracing techniques. In IR-based tracing, term similarity is used to identify related documents. Therefore, these approaches are not able to identify documents that are conceptually related, but that do not include similar terms. On the other hand, the main advantage of IR-based approaches is that they are fully automated.

Despite these limitations, our approach can still be very useful to maintainers. In fact, the goal of our work is not to replace maintainers, but to support them in the update of the requirements. Allowing maintainers to rapidly find the impacted requirements for most of the changes is expected to encourage them to keep the specification up-to-date. Maintainers should, however, keep in mind that the approach can also miss some of the impacted requirements in some cases. In such cases, if the maintainers are familiar with the specification, then they might spot that something is missing, and thus try to find it by doing an additional search. If the maintainers are not able to spot that an impacted requirement is missed, then this will lead to inconsistency in the requirements specification. Without our approach, however, the maintainer will have to find the impacted requirements fully manually, a task that requires much effort and is error prone. This is likely to lead to more inconsistency, or even discourage the maintainer from updating the specification at all. Therefore, we expect our approach to reduce the inconsistency problem, although the generated results are not 100% correct.

8.2. Heuristics validation

Our approach for identifying the commits that are likely to impact requirements was built based on the observations presented in Section 4.1. Therefore, we consider the evaluation of the approach as an initial validation of these heuristics. The heuristics can be divided in two categories: (1) heuristics about changes to be ignored and (2) heuristics about changes to be considered as relevant.

The heuristics about changes to be ignored (Observations 1, 3, and 4) introduce the risk that changes that impact the requirements are omitted during the analysis. In the evaluation, this risk did not materialize in the AquaLush and Connect projects. In the iTrust project, two relevant changes were missed because we ignored a change in a method body (Observation 1) and a change in the value of a constant, which we also did not consider as relevant in the implementation. Observation 6, which also introduces the risk that requirements-impacting changes are omitted, is not validated as it had no impact in our current implementation of the comparing tool.

Heuristics about changes to be considered (Observations 2 and 5) introduce the risk that code changes that only relate to implementation details are considered as relevant and thus overwhelm the user with irrelevant changes. This might also decrease the credibility of the tool to the user and thus discourage its use. The severity of this risk depends on the amount of irrelevant information that are detected. The risk is high when the tool has a low specificity and thus detects too many irrelevant commits as relevant. From the evaluation, we see that this risk is

under control as the specificity was above 80% for both AquaLush and iTrust. In the Connect project, for which the lowest specificity was obtained (69%), the main source of introducing irrelevant changes was the addition of new methods that did not relate to addition or extension of features (Observation 2).

Although the current evaluation gives a positive hint regarding the validation of the heuristics, a more in depth evaluation is still needed. In fact, the evaluation made on the three projects was primarily meant to evaluate the end-to-end approach, thus the insight it gives regarding the validity of the heuristics is limited.

8.3. Scope

Our approach can be used when an initial requirements specification exists but is not kept up-to-date because of time and cost constraints. For maintainers who think that updating requirements is only a waste of time and does not bring any benefits, our approach is useless. In fact, our approach is only meant to reduce the effort required to update requirements, but it does not eliminate the effort completely nor does it force the maintainer to update the requirements.

The current approach detects the requirements that are impacted by the code changes, but it does not detect missing requirements in the specification. However, it can still support the maintainer in adding new requirements by finding the existing requirements that are related to the new one, and thus help the maintainer decide how and where to add the new requirement in the specification.

The performance of the tracing part of the approach depends on the comments existing in the code. Therefore, our approach is expected to work much better for code that is well commented than for code that is uncommented or badly commented.

8.4. Threats to validity

Construct validity As we could not find requirements documents that include all the requirements for the Connect project, we used the document that includes the new requirements for release 3.3 only. Therefore, it is possible that there are other impacted requirements from other releases that we did not trace to in this experiment.
We consider this threat to be limited because if the approach succeeds in identifying the impacted requirement among the requirements for release 3.3, then the comparing tool succeeded in extracting terms that describe the change well enough that these same terms should also allow identifying impacted requirements in other releases.

Internal validity Defining ourselves the changes to be implemented for AquaLush introduces the threat of limiting the changes to those that our approach can detect. As the detection of a change is very dependent on how the change is implemented (e.g., is a new method added? Is only an existing method modified?), and as the implementation was carried out by an external developer, this threat is limited. Additionally, this threat does not appear in the iTrust and Connect projects, as, in these projects, we did not select the changes to be applied.
The classification of the issues in Jira for the Connect project is not necessarily accurate (e.g., a task does not necessarily mean that the change is requirements-affecting). This leads to the threat that some commits were misclassified. However, this threat is limited as the classification was carried out by the owners of the code, who are most familiar with the project. Furthermore, the classification is not subjective as we were not involved in it.

External validity Our approach is developed under the assumption that one commit should have a single purpose. Using multi-issue commits is likely to decrease the effectiveness of the approach. Although using single purposed commits might not be universal, it is a recommended practice for version control systems. For example, this is the second practice listed in the Apache's *Subversion Best Practices* [18]. It is also appears in version control tutorials and academic materials such as *the version control concepts and best practices* from Ernst [19].

The current approach was developed and tested on Java only. However, as object-oriented languages have many programming practices in common, we expect the approach to be applicable to other OO languages, while requiring minor modifications only.

9. RELATED WORK

Our related work section is composed of three parts. The first one is about requirements evolution, the second is about software traceability and its use for requirements evolution, and the third is about source code differencing techniques.

9.1. Requirements update and evolution

Managing the evolution of requirements is a problem that has been addressed by several researchers from the software engineering field. Hermann et al. [20] address the problem of specifying new requirements by proposing an approach to specify delta requirements in detail while describing the rest of the system on a higher-level of granularity. Zowghi and Gervasi [21] explore ways to ensure that the requirements specification is correct, consistent, and complete after each change using different validation checks. However, we are the first, to the best of our knowledge, to propose an automated approach for identifying the impacted requirements of a software system based on the changes applied to the code. What is also special in our work is that we assume that the code is changed before the requirements, whereas for most existing approaches for managing the evolution and update of requirements, the authors assume that the changes are applied at the requirements level first and then are propagated to the lower level artifacts and source code, as in [22].

Our work relates to co-evolution as it aims at supporting the co-evolution of the code and the requirements specification. Most of the existing co-evolution approaches address the co-evolution of the implementation with the design [23–25]. Reiss developed a constraint-based approach to ensure the consistency of different software artifacts. A number of rules that reflect the consistency between artifacts are defined and then used to detect inconsistencies between evolving artifacts [26]. Hammad *et al.* [27] propose an approach that automatically identifies whether a code change impacts the design. Their approach is applicable for design that is specified in the form of UML class diagrams. Our work is different for two reasons. First, we have to analyze a document that is written in natural language (the requirements). Second, unlike design and implementation, which both relate to the solution domain, the requirements relate to the problem domain, which makes the mapping between the requirements and the code more complex.

In [28], we propose an approach that uses the changes in high-order tests (such as acceptance tests) to identify impacted requirements. For this, a set of traceability links between the requirements and the acceptance tests are required. Using these links, we can trace back all modified tests to the requirements they derive from. The advantage of our current approach is that no tests or traceability links are required as the analysis is performed on the source code directly and is automatically traced back to the requirements.

9.2. Software traceability

Software traceability is one of the main approaches that are meant to support the maintenance of software artifacts. Research in the field of traceability is abundant and covers several tracing aspects such as the automatic generation of traceability links [29], the management of traces evolution [30], the automatic maintenance of traces [31], the use of traceability links to generate requirements views [32], and tools for generating traceability links and managing them [9, 33, 34].

For the automated generation of candidate traceability links, IR methods have been frequently used as they allow linking all types of artifacts that contain text to each other based on term similarity [35–38]. Several of the approaches also combine IR methods with other techniques such as machine learning [39], execution tracing [40], analyst feedback [41], refactoring [42],

ontologies [43, 44], co-occurrence of terms [45], and historical co-changes [46] to improve the tracing results.

The tracing used in our approach, which is based on IR, is different from the previous works. In fact, we only trace the code changes that are likely to impact the requirements. Additionally, we complement the terms used for the tracing by data from the call hierarchy of the changed elements.

Some other approaches for automated generation of candidate traceability links exist that are not based on IR methods. For example, Egyed [47] proposes a different tracing approach that is based on trace analysis. To run this approach, not only an initial set of manual traces is needed, but also a set of scenarios that will be executed against the code is required. Delater and Paech [48] developed an approach to link requirements and source code during software development by capturing links between the work items that developers are working on, the requirements they are looking at when implementing a work item and the code committed in the version control system for the same work item. Their automated approach is to be used during software development, in projects where a list of features exist and the implementation was broken down into planned work items that have been assigned to developers.

To the best of our knowledge, the tracing from code changes to requirements has not been addressed yet by existing traceability approaches, whether IR-based or not. Our work also goes beyond simple tracing by providing maintainers with concrete suggestions of what requirements are impacted when the code is changed.

9.3. Source code differencing

As detecting code changes is useful for supporting software comprehension and maintenance tasks, several source code differencing approaches were developed. Textual differencing, such as [49], is one of the earliest differencing techniques. It compares files line by line and detects the ones that changed. This technique has the advantage of identifying all the differences between two textual files. The main limitation of using textual differencing is that it includes too many changes that are likely to be irrelevant in the context of source code analysis such as changes in blank spaces and changes in the documentation. To filter out irrelevant change details, the code syntax was taken into consideration for several differencing techniques (e.g., [50]). There are also several approaches that identify semantic changes in the code. The tool from Jackson and Ladd [51] detects the semantic differences in programs based on the observable input–output behavior of procedures. Difference Extactor (Dex), which is a tool for analyzing semantic and syntactic changes in large code bases, is proposed as a means to collect information about the nature of code changes in software code projects [52]. Chianti [53] is a static analysis tool that analyzes the differences between two versions of Java code and identifies the tests (unit and regression) that are impacted by the changes. It also identifies, for each impacted test, the set of changes that were responsible for the change of the test's behavior. The JDiff tool implemented by Apiwattanapong et al. [54] uses a graph representation of the code to identify and classify changes at the statement level between two versions of object-oriented code. Symdiff [55] is a semantic differencing tool for imperative programs. Its main property is that is operates at an intermediate verification language, which makes the core analysis algorithm applicable to several languages.

Our differencing approach is different from the reviewed ones as it only identifies the changes that are likely to impact the requirements and ignores changes that are bug fixes or refactoring. It is also very simple and can be easily adapted by the user to the characteristics of the explored project. Additionally, in this work, the goal is not to detect changes in source code, but to gather as much relevant information as possible about the context of the change. This is carried out, for example, by inspecting the call hierarchy and the parent elements of the changed parts.

10. CONCLUSION AND FUTURE WORK

In this article, we presented an approach for identifying impacted requirements based on source code changes. Our approach is meant to support maintainers in the update of the requirements specification by automatically identifying the parts that are likely to be impacted after each code commit. The

approach is composed of three main steps. First, the old and new code are compared with each other in order to identify if there are requirements-impacting changes. If such changes are detected, then terms describing the change are extracted from the code and are traced to the requirements in order to identify the parts that are likely to be impacted. Finally, the impacted requirements are displayed to the maintainer, who can then update them and save the changes. To evaluate our approach, we developed a set of tools that we used to run the approach on three case studies, namely, AquaLush, iTrust, and Connect. Our approach succeeded to identify between 70% and 100% of the impacted requirements while filtering out more that 80% of the non-impacted requirements in the specification. Automatically identifying the requirements that are likely to be impacted after each source code change is expected to reduce the time and effort needed for updating requirements. Thus, it should also encourage maintainers to regularly update the specification.

For future work, there are two main directions we would like to explore. The first direction aims at improving the tracing approach. We would like to do so by extending the approach so that weights are given to the keywords used for the tracing. The weights will depend on the source of the keyword, so that keywords extracted from an added method get a higher weight than those obtained from the call hierarchy. The tracing can also be improved by using more elaborate tracing techniques, which combine IR with machine learning or analyst feedback.

The second direction for the future work relates to evaluating the usefulness of the approach for maintainers. For this, we plan to conduct a controlled experiment where we compare the time needed to do the maintenance task, the correctness of the update, and the confidence of the maintainer about it when using and when not using our approach. To conduct such an experiment, we need a user-friendly version of the tool that nicely displays the requirements that are likely to be impacted to the maintainer.

ACKNOWLEDGMENTS

This work was partially funded by the Swiss National Science Foundation under grant PDFMP2-122969.

REFERENCES

1. Bennett Keith H, Rajlich Václav T. Software maintenance and evolution: a roadmap. In The Future of Software Engineering, Finkelstein A, Kramer J (eds.). 2000; 73–87.
2. Lethbridge Timothy, Singer Janice, Forward Andrew. How software engineers use documentation: The state of the practice. *IEEE Software* 2003; **20**(6):35–39.
3. Gorschek Tony, Svahnberg Mikael. Requirements experience in practice: Studies of six companies. In Engineering and Managing Software Requirements, 2005; 405–424.
4. Glinz Martin. On non-functional requirements. In *15th IEEE International Requirements Engineering Conference* (*RE'07*), 2007; 21–26.
5. Ben Charrada Eya, Koziolek Anne, Glinz Martin. Identifying outdated requirements based on source code changes. In *20th IEEE International Requirements Engineering Conference* (*RE 2012*), 61–70, 2012.
6. Duc Anh Nguyen, Cruzes Daniela S, Ayala Claudia, Conradi Reidar. Impact of stakeholder type and collaboration on issue resolution time in OSS projects. In Open Source Systems: Grounding Research, Springer: 2011; 1–16.
7. Glinz Martin. Rethinking the notion of non-functional requirements. In *Third World Congress for Software Quality*, 2005; 55–64.
8. Levenshtein Vladimir I. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, 1966; **10**:707–710.
9. Huffman Hayes Jane, Dekhtyar Alex, Sundaram Senthil Karthikeyan, Holbrook Elizabeth Ashlee, Vadlamudi Sravanthi, April Alain. Requirements TRacing On target (RETRO): improving software maintenance through traceability recovery. *Innovations in Systems and Software Engineering* 2007; **3**(3):193–202
10. Doar Matthew. JDiff – what really changed? *Java Developer's Journal*, 2002.
11. Basili Victor R. Software modeling and measurement: The goal/question/metric paradigm. Technical Report CS-TR-2956, UMIACS-TR-92-96, University of Maryland, September 1992.
12. Olson David L, Delen Dursun. Advanced Data Mining Techniques. Springer: 2008.
13. Dominich Sándor. The Modern Algebra of Information Retrieval, Springer: 2008.
14. Fox Christopher. Introduction to Software Engineering Design: Processes, Principles and Patterns with UML2. Addison Wesley, 2006.
15. Ben Charrada Eya, Caspar David, Jeanneret Cédric, Glinz Martin. Towards a benchmark for traceability. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2011)*, 21–30, 2011.
16. Meneely Andy, Smith Ben, Williams Laurie. Appendix B: iTrust electronic health care system case study. In Software and Systems Traceability. 2012.

17. Williams Laurie, Xie Tao, Meneely Andy, Hayward Lauren, King Jason. iTrust medical care requirements specification. Versions of September 3rd, 2010: agile.csc.ncsu.edu/iTrust/wikidoku.php?id=requirements&rev=1283530873 and of February 7th, 2011: rev=1297120633.
18. Apache. Subversion best practices. (Available from: http://svn.apache.org/repos/asf/subversion/trunk/doc/user/svn-best-practices.html.) [8 May 2014].
19. Ernst Michael. Version control concepts and best practices. (Available from: http://homes.cs.washington.edu/~mernst/advice/version-control.html.) [8 May 2014].
20. Herrmann Andrea, Wallnöfer Armin, Paech Barbara. Specifying changes only–a case study on delta requirements. *Requirements Engineering: Foundation for Software Quality* (REFSQ 2009) 2009; 45–58.
21. Zowghi Didar, Gervasi Vincenzo. On the interplay between consistency, completeness, and correctness in requirements evolution. *Information and Software Technology* 2003; **45**(14):993–1009.
22. Ernst Neil A, Borgida Alexander, Mylopoulos John. Requirements evolution drives software evolution. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2011)*, 16–20, 2011.
23. Mens Kim, Kellens Andy, Pluquet Frédéric, Wuyts Roel. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures* 2006; **32**(2–3):140–156.
24. Cazzola Walter, Pini Sonia, Ghoneim Ahmed, Saake Gunter. Co-evolving application code and design models by exploiting meta-data. In *2007 ACM Symposium on Applied Computing* (*SAC '07*), 2007; 1275–1279.
25. D'Hondt Theo, De Volder Kris, Mens Kim, Wuyts Roel. Co-evolution of object-oriented software design and implementation. In Software Architectures and Component Technology, Springer: 2002; 207–224.
26. Reiss Steven P. Incremental maintenance of software artifacts. *IEEE Transactions on Software Engineering* 2006; **32**(9):682–697.
27. Hammad Maen, Collard Michael L, Maletic Jonathan I. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Journal* 2011; **19**(1):35–64.
28. Ben Charrada Eya, Glinz Martin. An automated hint generation approach for supporting the evolution of requirements specifications. In *Joint ERCIM Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL 2010)*, 58–62, 2010.
29. Antoniol Giuliano, Canfora Gerardo, Casazza Gerardo, De Lucia Andrea. Information retrieval models for recovering traceability links between code and documentation. In *International Conference on Software Maintenance*, ICSM '00, 2000; 40–49.
30. Cleland-Huang Jane, Chang Carl K, Christensen Mark. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering* 2003; **29**(9):796–810.
31. Mäder Patrick, Gotel Orlena. Towards automated traceability maintenance. *Journal of Systems and Software* 2012; **85**(10):2205–2227.
32. Lormans Marco, Deursen Arie, Gross Hans-Gerhard. An industrial case study in reconstructing requirements views. *Empirical Software Engineering* 2008; **13**(6):727–760.
33. Lin Jun, Lin Chan Chou, Cleland-Huang Jane, Settimi Raffaella, Amaya Joseph, Bedford Grace, Berenbach Brian, Ben Khadra Oussama, Duan Chuan, Zou Xuchang. Poirot: A distributed tool supporting enterprise-wide automated traceability. In *IEEE International Requirements Engineering Conference (RE'06)*. 2006; 363–364.
34. De Lucia Andrea, Fasano Fausto, Oliveto Rocco, Tortora Genoveffa. Fine-grained management of software artefacts: the ADAMS system. *Software: Practice and Experience* 2010; **40**(11):1007–1034.
35. Antoniol Giuliano, Canfora Gerardo, Casazza Gerardo, De Lucia Andrea, Merlo Ettore. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 2002; **28**(10):970–983.
36. Huffman Hayes Jane, Dekhtyar Alex, Sundaram Senthil Karthikeyan. Advancing candidate link generation for requirements tracing: the study of methods. *IEEE Transactions on Software Engineering* 2006; **32**(1):4–19.
37. Marcus Andrian, Maletic Jonathan I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th International Conference on Software Engineering* (*ICSE '03*). IEEE Computer Society: 2003; 125–135.
38. De Lucia Andrea, Fasano Fausto, Oliveto Rocco, Tortora Genoveffa. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2007; **16**(4):13.
39. Cleland-Huang Jane, Czauderna Adam, Gibiec Marek, Emenecker John. A machine learning approach for tracing regulatory codes to product specific requirements. In *32nd ACM/IEEE International Conference on Software Engineering*, ICSE '10, 155–164. ACM, 2010.
40. Eaddy Marc, Aho Alfred V., Antoniol Giuliano, Guéhéneuc Yann-Gaël. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *16th IEEE International Conference on Program Comprehension*, June 2008; 53 –62.
41. Huffman Hayes Jane, Dekhtyar Alex, Sundaram Senthil Karthikeyan. Improving after-the-fact tracing and mapping: supporting software quality predictions. *IEEE Software* 2005; **22**(6):30–37.
42. Mahmoud Anas, Niu Nan. Supporting requirements traceability through refactoring. In 21*st IEEE International Requirements Engineering Conference* (*RE 2013*). IEEE: 2013; 32–41.
43. Li Yonghua, Cleland-Huang Jane. Ontology-based trace retrieval. In *International Workshop on Traceability in Emerging Forms of Software Engineering* (*TEFSE*), IEEE. 2013; 30–36.

44. Assawamekin Namfon, Sunetnanta Thanwadee, Pluempitiwiriyawej Charnyote. Ontology-based multiperspective requirements traceability framework. *Knowledge and Information Systems* 2010; **25**(3):493–522.
45. Dasgupta Tathagata, Grechanik Mark, Moritz Evan, Dit Bogdan, Poshyvanyk Denys. Enhancing software traceability by automatically expanding corpora with relevant documentation. In *29th IEEE International Conference on Software Maintenance* (*ICSM 2013*), Sept 2013, 320–329.
46. Ali Nasir, Jaafar Fehmi, Hassan Ahmed E. Leveraging historical co-change information for requirements traceability. In *20th Working Conference on Reverse Engineering*, *WCRE 2013*; 361–370, Oct 2013.
47. Egyed Alexander. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering* 2003; **29**(2):116–132.
48. Delater Alexander, Paech Barbara. Tracing requirements and source code during software development: An empirical study. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Oct 2013; 25–34.
49. Hunt James Wayne, McIlroy M Douglas. An algorithm for differential file comparison. Bell Laboratories 1976.
50. Yang Wuu. Identifying syntactic differences between two programs. *Software: Practice and Experience* 1991; **21**(7): 739–755.
51. Jackson Daniel, Ladd David A. Semantic diff: a tool for summarizing the effects of modifications. In *International Conference on Software Maintenance*. 1994; 243–252.
52. Raghavan Shruti, Rohana Rosanne, Leon David, Podgurski Andy, Vinay Augustine. Dex: A semantic-graph differencing tool for studying changes in large code bases. In *20th IEEE International Conference on Software Maintenance*. IEEE: 2004; 188–197.
53. Ren Xiaoxia, Shah Fenil, Tip Frank, Ryder Barbara G, Chesley Ophelia. Chianti: a tool for change impact analysis of Java programs. In ACM Sigplan Notices. ACM: 2004; **39**:432–448.
54. Apiwattanapong Taweesup, Orso Alessandro, Harrold Mary Jean. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering* 2007; **14**(1):3–36.
55. Lahiri Shuvendu K, Hawblitzel Chris, Kawaguchi Ming, Rebêlo Henrique. Symdiff: A language-agnostic semantic diff tool for imperative programs. In Madhusudan P, Seshia SanjitA (eds.) *Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, Springer, 2012; 712–717.